Multigrain Affinity for Heterogeneous Work Stealing

Jean-Yves Vet, Patrick Carribault, Albert Cohen

Multigrain Affinity for Heterogeneous Work Stealing

Jean-Yves Vet[1], Patrick Carribault[1], Albert Cohen[2]

[1] CEA, DAM, DIF, F-91297 Arpajon, France
[2] INRIA and École Normale Supérieure, Paris, France

Abstract. In a parallel computing context, peak performance is hard to reach with irregular applications such as sparse linear algebra operations. It requires dynamic adjustments to automatically balance the workload between several processors. The problem becomes even more complicated when an architecture contains processing units with radically different computing capabilities. We present a hierarchical scheduling scheme designed to harness several CPUs and a GPU. It is built on a two-level work stealing mechanism tightly coupled to a software-managed cache. We show that our approach is well suited to dynamically control heterogeneous architectures, while taking advantage of a reduction of data transfers.

Keywords: Heterogeneous Computing, Work Stealing, Software Cache, Sparse LU Factorization, GPGPU.

1 Introduction

With the advent of the multicore era, processor architectures evolve to include more processing units either by increasing the width of each functionnal unit (e.g., the new AVX instruction set) or by replicating simpler cores and factoring some of their resources (e.g., NVidia's GPUs). These two directions are currently exploited in High-Performance Computing. As of June 2011, several of the clusters within the 10 most powerful supercomputers listed in the Top500 are heterogeneous. Therefore, harnessing both processor architectures (CPUs and GPUs) is mandatory to reach high performance. When scaling computations to such heterogeneous architectures, data management is a highly sensible parameter and a key to the efficient exploitation of the computing resources. This paper introduces a new scheduling technique coupled with a software cache for locality optimization to exploit both CPUs and GPUs in the context of irregular numerical computations.

It is organized as follows. Section 2 introduces our motivating example. Section 3 shows the necessity of a multigrain mechanism when tasks are executed by heterogeneous processing units. Then, Section 4 describes our software cache. Performance results are presented in Section 5. Finally, Section 6 exposes related work before concluding in Section 7.

Fifth Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG '12), in conjunction with the 7th Int. Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC '12), January 2012, Paris, France

2 Motivating Example: Sparse LU Factorization

The performance of Block LU factorization depends on careful data management and work scheduling. The algorithm iterates over a matrix A to decompose it into a product of a lower triangular matrix L and an upper one U such as $A = LU$. Each iteration consists of three interdependent steps (**Figure 1a**). Within each step, multiple operations can be launched in parallel and grouped into tasks. For large matrices, the collection of block operations at step 3 dominates computation time. This particular step runs block multiplications that can be implemented as highly optimized Basic Linear Algebra Subprograms (BLAS) such as CUBLAS for a NVIDIA GPU or the Intel Math Kernel Libray (MKL) for CPUs. We adopt here a right-looking LU implementation since it offers a high amount of data-parallelism in step 3.

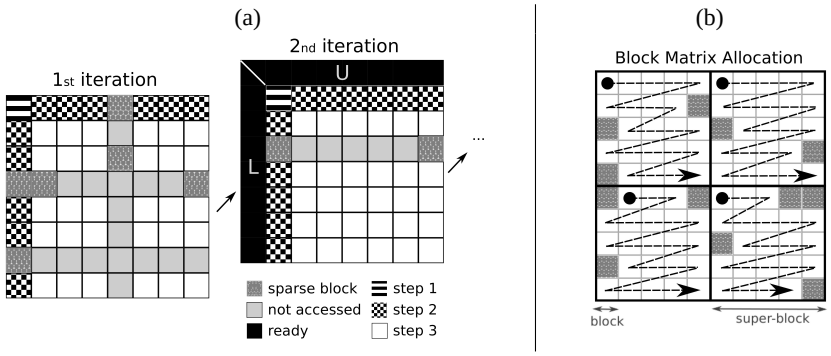

Fig. 1. (a) Data modified by each step during the first and second iteration of a sparse LU decomposition. **(b)** Allocation of a sparse block matrix using super-blocks. Each super-block contains 5×5 blocks that are empty or full.

Studies reveals that computing dense LU factorization on GPUs may lead to significant speedups thanks to level-3 CUBLAS kernels [23, 8]. Stomov et al. proposed new techniques applied to heterogeneous dense LU factorization leading to a balanced use of a multicore CPU and a GPU [22]. They designed a new algorithm which aims at reducing the amount of pivoting. Then, they empirically found the best static schedule to reach good performance. This predefined scheduling performs well since the workload is known for each iteration. The problem is less predictable when the matrix contains sparse blocks, motivating the search for dynamic scheduling algorithms. Deisher et al. implemented a dynamic load balancing scheme to take advantage of CPUs and an Intel MIC (Many Integrated Core) architecture but it was also tailor-made for a dense LU factorization [11].

Sparse direct solvers emerged in the past ten years for shared memory machines, distributes memory platforms, or combination of the two (MUMPS [3], PaStiX [13], SuperLU [20], ...). To the best of our knowledge, the efficient execution of sparse LU factorization harnessing multiple CPUs and a GPU has not received much attention. The sparseness of the matrix has a huge impact on workload distribution. Sparse blocks may turn into dense blocks from one iteration to another. Thus, the workload of a task may change at any time. Obviously, a static approach or a dynamic schedule based

on cost models could suffer from severe imbalance. Based on this observation, we designed mechanisms to dynamically guide the cooperation between all heterogeneous processing units. We employ a sparse LU decomposition without pivoting to evaluate our scheduler. The goal of this paper is not to propose a new direct solver for sparse matrices. Nonetheless, some statistical techniques could be used to ensure the stability of this algorithm. For instance, the Partial Random Butterfly Transformation (PRBT) presented by Baboulin and al., is designed to avoid pivoting in LU factorization while getting an accuracy close to the partial pivoting solution [6].

3 Task Granularity

When a program is decomposed into tasks, the quantity of operations contained in each task has a major effect on performance. Granularity has obviously a direct impact on the number of tasks, but it also modifies the way processing units are harnessed. For instance, GPUs are made of several hundreds of light cores called Stream Processors (SP). A task would intrinsically require a high degree of parallelism to properly benefit from that massively parallel architecture. Since a task may be either processed by CPU core or a GPU, it implies trade-offs on granularity.

In the following, the input matrix consists of N^2 blocks of size 192×192. N varies from 40 to 160, hence the total number of full and empty blocks varies from $1,600$ to $25,600$ (from 256 Mo to 3,775 Go if we consider dense matrices). One third of the blocks located off-diagonal are sparse. The position of each empty block is determined in advance for a given matrix size, so that the average performance between several executions can be computed. Values inside dense blocks are randomly generated. We believe that the choice of the block size is a good compromise. One the one hand, we wish the block size as small as possible to better represent the sparsity of a matrix. For instance, if just one value in a block is non-null, the whole block can not be mapped as empty and computations linked to that block can not be avoided. A small block size also permits to extract more fine-grained parallelism and thus, brings more flexibility for the scheduler. On the other hand, we want a large enough dimension to preserve good performance on block operations. In **Figure 2a** and **2b**, we show performance of blocks multiplications on both GPU and CPU versus different block dimensions.

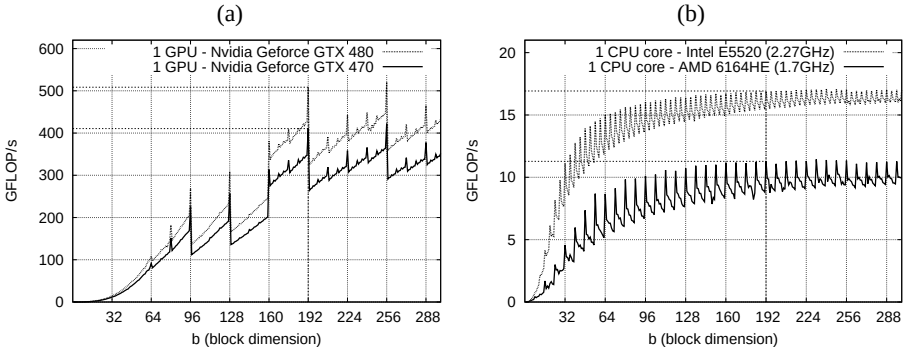

Fig. 2. Block multiplication performance versus block size (average on 100 runs), $C = AB + C$ where A, B and C are b×b blocks: **(a)** Nvidia CUBLAS 4.0 - 20 SGEMM streamed at the same time (data transfers are not taken into account). **(b)** Intel MKL 10.2.6 - SGEMM.

In this section, we first evaluate the impact of different task granularites on CPUs and a GPU. We present the outcome of specific experiments with the third step of our sparse LU implementation with a shared memory SMP machine hosting two AMD 6164HE (2x12 CPU cores) and a Nvidia Geforce GTX 470 (448 SPs). Then, we explain our multigrain task mechanism.

3.1 Single Granularity

In this evaluation, our runtime creates 24 threads to operate 1 task producer, 22 CPU consumers and 1 GPU consumer. Each thread is bound to a CPU core so that several workers cannot be multiplexed on the same core. Tasks are first produced and shared among all consumers in a round robin way (**Figure 3, A1**). As soon as a CPU consumer completes all its tasks, it tries to steal work from another CPU worker (**Figure 3, A2**). Similarly, when the producer reaches a synchronization barrier, it acts as a worker and randomly steals tasks to help CPU workers instead of waiting (**Figure 3, A3**). Thus, load balancing is automatically managed by work distribution and work stealing [10, 1] mechanisms. When a task is scheduled on the GPU worker (**Figure 3, B1**), data are copied into device memory, computed and finally transferred back to host memory. For some evaluations we desactivated a few consumers to adjust the runtime to pure CPUs or pure GPU executions. Thus, pure CPU execution implies 22 workers and 1 producer using 23 cores, whereas pure GPU execution means 1 worker attached to 1 core dedicated to pilot the GPU.

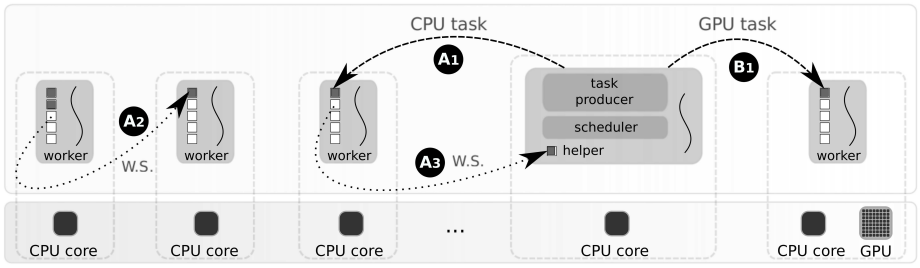

Fig. 3. Scheduling with single task granularity.

Our first evaluation studies the performance of fine-grained task scheduling. In this case, each task modifies only one 192×192 matrix block. This block size is small enough for a sparse matrix representation while associating sufficient quantity of operations per task to amortize the (low) task spawning and scheduling overhead. In our second evaluation, each task updates several matrix blocks. We call this group of blocks a *super-block*. Since multiple blocks are accessed by the same processor, we change the way data are allocated to increase spatial locality. Thus, we allocate each super-block independently as shown in **Figure 1b**. All data are allocated in page-locked mode to prevent the Cuda runtime from managing unnecessary copies from pageable to non-pageable memory. To determine the size of each super-block in advance, we pre-detect which blocks will stay empty prior to launching the LU decomposition. This facilitates

data transfers to GPU memory. The execution time of this pre-detection step is negligible. We use CUBLAS functions (CUDA Toolkit 4.0) in our GPU tasks; but kernels could also be automatically generated by means of directives [12] or by automatically parallelizing loops that can benefit form GPU execution [19]. We choose super-blocks consisting of 5×5 blocks, because it contains (on average) sufficient full blocks to benefit from their spatial locality and trigger packed transfers via the PCI-Express. It also allows us to increase GPU occupancy by launching several kernels on enough CUDA streams. In other words, several functions are started on different matrix blocks simultaneously.

Figure 4a compares the performance of the two approaches. We notice that fine granularity enhances performance on CPUs, especially with smaller matrices: it allows to create more tasks, improving load balancing opportinities between workers. We observe a different behavior for the GPU worker which achieves better performance with coarse-grained tasks: each task contains more data which makes data transfers more efficient. Whereas the same amount of data is send over the PCI-Express bus, less transfers are issued, resulting in reduced overhead and increased performance. In both cases GPU performance are lower than those presented in **Figure 2a**. This is mainly due to data transfers.

We can only hope that heterogeneous, multigrain task scheduling combines the advantages of both fine-grained and coarse-grained configurations. In the next section we describe our two-level task granularity mechanism.

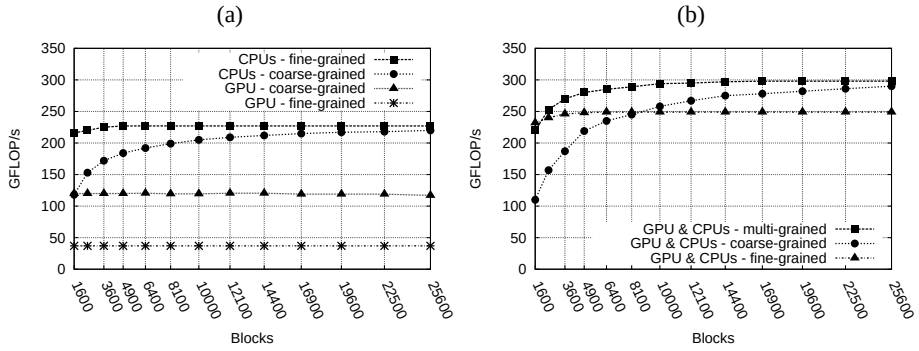

Fig. 4. Sparse LU step 3, single precision, average on 10 executions for each result: **GPU**: Nvidia Geforce GTX 470 controled by one 1 worker binded on 1 CPU core (AMD 6164HE). **CPUs**: consit of 23 workers binded on 23 CPU cores. **(a)** homogeneous **(b)** heterogeneous scheduling.

3.2 Multi Granularity

The challenge is twofold. The runtime part has to handle the two granularities for a specific task, while maintaining a dynamic behavior to adjust workload between heterogeneous processors. It is important to notice that CPUs and GPUs are not controlled the same way. A GPU is managed via one single system thread, while a multicore CPU requires at least one thread per core. It commonly leads to competition between the GPU and each CPU workers. Theoretical peak performance of a recent GPU is much higher than the one of a CPU core released around the same period. Task consumption

is then significantly unbalanced. Several schedulers compensate this difference in processing power by introducing cost models [7]. Unfortunately, cost models are not ideal to handle sparse codes due to workload variations.

To better balance the workload, we virtually pack several workers to build groups of relatively close processing power. This also brings together processors sharing the same affinity for a particular task granularity. Thereby, we gather processing power of two multicore CPUs into one group. This group which contains all CPU workers competes now with the GPU worker. To integrate this new level of scheduling into our runtime, we define *super-tasks* as sets of tasks operating on the same super-block. Spatial locality is maintained within a super-block, allowing the GPU worker to trigger packed transfers through the PCI-Express bus.

A super-task is pushed in a double-ended queue (deque) when it is produced (**Figure 5, A**). Only one worker at a time of each group is allowed to pop a super-task. CPU workers may dequeue super-tasks only from one end (**Figure 5, B1**), whereas the GPU worker can only dequeue from the other end (**Figure 5, C1**). As soon as a CPU worker picks a super-task, it breaks it down and generates smaller tasks that are shared between CPU consumers (**Figure 5, B2**). Conversely, when the GPU worker picks a super-task, all inner tasks are either launched from one function call or several streamed calls to maximize SPs occupancy.

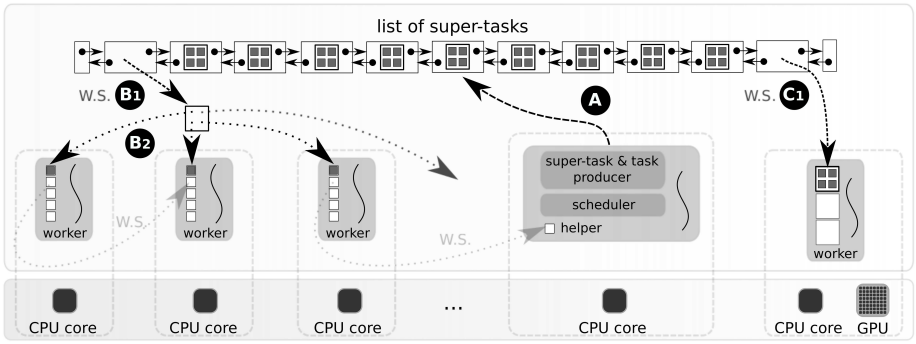

Fig. 5. Super-task scheduling.

Work stealing between CPU workers described in **Figure 3** is maintained. Thus, we obtain a two-level task scheduler. Super-task picking sets up an efficient load balancing between heterogeneous processors, while fine-grained stealing equilibrates tasks consumption between processing units of the same kind. When the task queue of a CPU consumer becomes empty, the worker tries to steal a task in another CPU worker queue. If this operation does not succeed, the CPU worker tries then to pick a super-task.

We now present the performance obtained by harnessing GPU and CPUs concurrently on the third step of our Sparse LU code. In our multi-grained version, a super-task operates on super-blocks, whereas a task works on blocks granularity. **Figure 4b** shows results of multi-grained and single-grained tasks scheduling. Multi-grained version reaches the best performance most of the time except for relatively small matrices where a fine-grained decomposition is slightly better.

4 Guided Software Cache

We showed that multigrain scheduling combines advantages from both coarse-grained and fine-grained decompositions. More specifically, like coarse-grained scheduling it increases spatial locality. In this section we focus on temporal locality improvements.

4.1 Design and Tuning

We designed a software cache to maximize data reuse in GPU memory, minimizing data transfers between host and device. It automatically allocates memory and triggers transfers when data need to be accessed by the host or when all data does not fit in device memory. Thus, it allows us to set up a dynamic data management, alleviating programming efforts.

The replacement policy is the key component of our software cache. To reduce data transfers over the PCI-express interconnect, the cache maintains the most relevant data in device memory. Each new piece of data that enters the cache is associated with a marker indicating its reuse potential during the whole program execution. Data linked to low reuse potential indices are flushed and transferred back first when free space is required. Each index can be provided either by the programmer for a better control, or by a preliminary step that profiles the first execution.

To tune the design of our software cache, we ran tests on the third step of the sparse LU code. From an iteration to another, the LU decomposition progresses along the diagonal of the matrix (**Figure 1a**). Super-blocks close to the bottom right corner are more used than the other ones. To symbolize this reuse potential, all super-blocks are tagged with the marker $m = i + j$ where i and j are their position indexes in the matrix. Data can also be locked in the software cache for a given period. For instance, a block computed during the second step can be required to update several super-blocks in the third step. If this block is not maintained in device memory, unnecessary transfers may me triggered. We force data to stay in the software cache by temporarily attributing a very high reuse value.

We conducted experiments with an Nvidia GeForce GTX 470. Up to 1GB are dedicated to the software cache, which is about 80% of the total amount of device GDDR5 memory. The rest seems to be needed by the Cuda runtime since we were not able to allocate a higher amount. We can clearly observe that the software cache reduces the number of memory transfers and total transfer time, due to a better exploitation of data locality (**Figure 6a**). A peak is reached with a square matrix containing $6,400$ blocks where our software cache brings up to a 207% performance improvement (**Figure 6c**). In this configuration, the $CT = \frac{computation\ workload}{transfers}$ ratio is maximized and most of the data can fit in device memory. Then, performance decrease with bigger matrices since all data cannot be maintained into GDDR5 memory, inducing more transfers. With small matrices, even if the cache is not fully utilized, data have a lower reuse potential due to the nature of the algorithm, leading to a less impressive CT ratio than with a $6,400$-block matrix (**Figure 6b**).

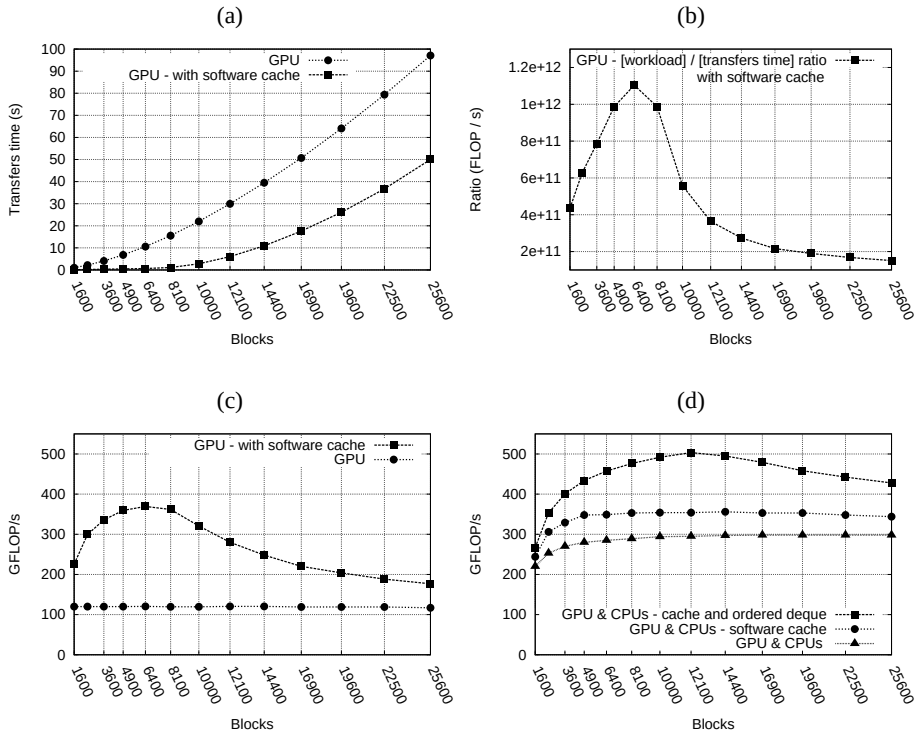

Fig. 6. Sparse LU step 3, single precision, average on 10 executions for each result: **(a)** cumulative time of all data transfers **(b)** ratio of workload to cumulative time of all transfers **(c)** GPU performance with and without software cache **(d)** heterogeneous execution and impact of the scheduler/cache interactions.

4.2 Interaction with Heterogeneous Scheduling

Heterogeneous scheduling and software cache performance are tightly linked. E.g., if a CPU worker picks a super-task associated with highly reusable data, it is very likely that last up-to-date copies reside in the device memory. Consequently, data should be flushed from the software cache to ensure coherency. We can infer that improper super-task picking may lead to a substantial increase of data transfers. To tackle with this problem, we modified the super-task deque to make it aware of the cache policy. When a super-task is created, it is also associated with a data reuse index and inserted in the deque which is now ordered by reuse potential. The GPU worker picks super-tasks with the highest index (**Figure 5, C1**), whereas CPU workers are restricted to the one with the lowest index (**Figure 5, B1**). This affinity mechanism guides super-task picking, minimizing undesired effects on the software cache and thus maximizing temporal locality. Now, our runtime benefits from both spatial and temporal locality thanks to the cooperation between our software cache and our heterogeneous scheduling.

Without modifying the way super-tasks are picked by workers, our software cache improves heterogeneous execution performance by up to 20% (**Figure 6d**). The sched-

uler also has a strong impact on the software cache efficiency: task scheduling guided by cache affinity brings up to 70% performance improvement compared to the version without software cache.

5 Integrated Performance Evaluation

We now evaluate the complete sparse LU algorithm (all steps) on two different systems. The first one is the previous AMD platform used to gather intermediate results. The second system is composed of two Intel Xeon E5520 (4 cores each) and an Nvidia GeForce GTX 480.

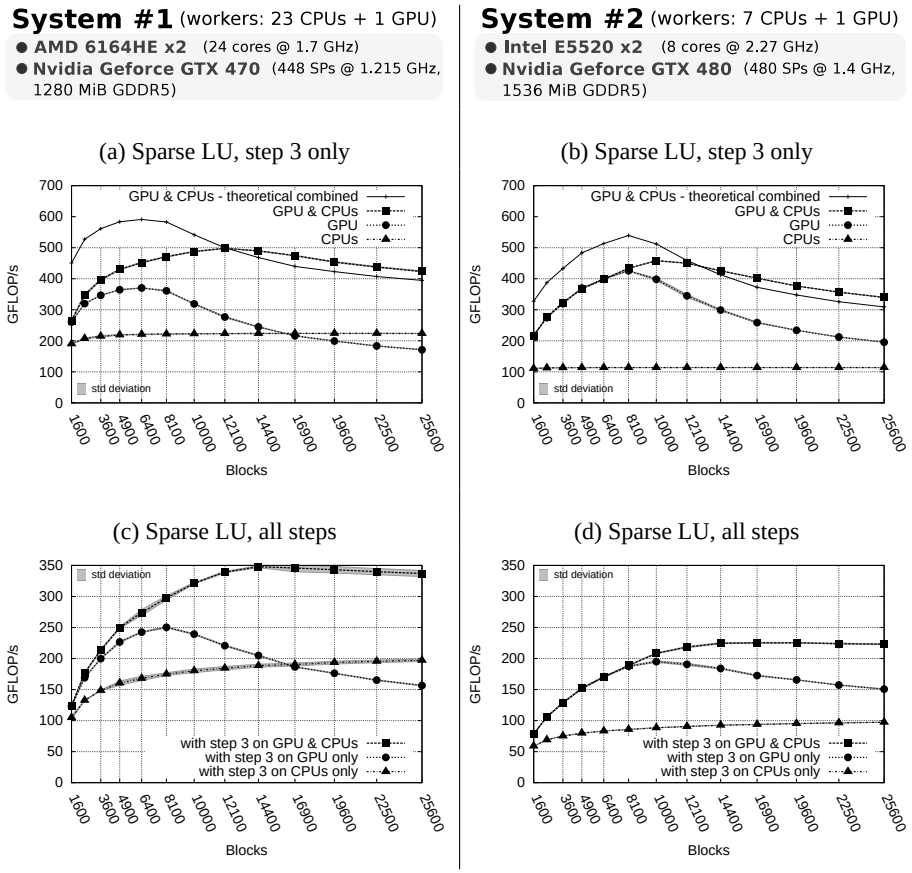

Fig. 7. Average on 25 executions for each result in single precision (the largest relative standard deviation is 2.25%): **(a)(b)** Step 3 only **(c)(d)** Global performance (step 3: GPU+CPUs, step 1&2: CPUs only).

We show that the scheduler has a strong impact on the software cache efficiency (**Figure 6d**). Indeed, the improvement is more impressive with a task scheduling guided by cache affinity. It brings up to 70% performance improvement compared to the version without a software cache. **Figure 7a** and **7b** show the performance of step 3 in our Sparse LU implementation. We notice that the performance of pure CPUs execution is about twice better on the first system, mainly due to a higher number of CPU cores. As expected, pure GPU executions reveal that the code runs globally faster on a Geforce GTX 480 compared to a Geforce GTX 470. The performance increase is not only induced by a higher computation power but it is also due to a larger amount of GDDR5 memory which enhances the impact of our software cache.

It is interesting to notice that heterogeneous executions outperform the cumulative performance of the GPU-only and CPU-only versions for large enough matrices. This strong result is due to the locality-aware policy of the hierarchical scheduler. Guided by data affinity, the scheduler encourages cooperation between CPUs and GPU. Since the GPU gets a higher affinity for particular tasks, it focuses on a smaller set of blocks, improving data locality. CPU workers preferentially attract tasks linked to the other data blocks, minimizing the need for coherence transfers. **Figure 7c** and **7d** show the overall performance of our Sparse LU implementation where the GPU only contributes during step 3.

6 Related Work

We discuss work related to task scheduling and to the management of deported data.

Quintin et al. present an hierarchical work-stealing mechanism designed to reduce the amount of communications in a distributed environment [21]. Even if the targeted platform is different, our multigrain scheduling presents some similitudes. They define groups of workers restricted to a single or a set of multicore CPUs. Within each group, a leader is designed to manage the workload and steal tasks with a large amount of work from other groups. The leader can also split a coarse grained task into smaller tasks to increase the amount of parallelism inside its group. The other workers perform the classical work-stealing algorithm inside their group.

Jiménez et al. propose a predictive task scheduler [18]. It is based on past performance history to improve load balancing between a CPU and a GPU, but data are not handled via a software cache and transfers should be explicitly managed by the programmer. Ayguadé et al. extend StarSs to support multiple CPUs and GPUs [5]. Task creation is made easier for programmers since it includes a source-to-source scheme designed to translate OpenMP-like pragmas into task definitions. It is also associated with a runtime system which schedules tasks according to their data dependences.

Performance models are also popular. Weights are for instance attached to a directed acyclic graph of tasks in order to adjust task affinities with processing units [7]. Such a scheduling strategy can be activated in StarPU [4], a runtime system designed to harness heterogeneous architectures within a SMP machine. It provides the programmer with an interface to encapsulate existing functions in task abstractions named “codelets”. Data transfers are then managed automatically by the runtime system. It globaly leads to good performance on dense linear agebra [2]. A similar approach is adopted by an

extension of the Kaapi runtime [14, 15] to schedule tasks on GPUs and CPUs [17]. It focuses more specifically on an affinity scheme built to improve the efficiency of heterogeneous work stealing. A task dependence graph (TDG) is partitioned and a two-level scheduling strategy is adopted. Partitions are distributed to the different processing units and the workload is balanced by using a locality guided work stealing to move some partitions. As for StarPU, deported data are maintained in a software-managed cache and coherency is ensured via a distributed shared memory (DSM). Unfortunately, evictions cannot be controlled, which may lead to more data transfers depending on the application.

The Cell BE version of the StarSs platform [9], also takes advantage of a two level scheduler. Tasks are bundled from a TDG according to data locality. A software managed cache is used for each Synergistic Processor Element (SPE) to reduce DMA transfers between a local storage and the main memory. The software cache employs a Least Recently Used policy (LRU) and spatial locality is enhanced through locality hints maintained by the runtime. This solution seems convenient for programmers since tasks are defined via pragmas, and reuse opportunities are adjusted runtime. On the other hand, tasks are bundled from a partial TDG and temporal locality may be underexploited when the whole program is considered. In addition, load balancing between heterogeneous units is less tedious with a Cell BE architecture. In CellSs, the Power Processing Element (PPM) can directly help the SPEs by stealing tasks. In the case of multiple CPU cores and a GPU, the need of quite different degrees of parallelism and the large disparity in processing power induce stronger constraints on task granularity.

Gelado et al. present GMAC, an asymmetric DSM designed to reduce the coherence traffic and make data management easier to program with heterogeneous systems [16]. Pages are protected on the system side and page faults are used to eagerly transfer data. However, the overhead of page faults may be avoided with task level parallelism by associating data with in/out information (e.g. StarPU codelets or StarSs pragmas).

7 Conclusions and Future Work

We presented mechanisms designed to harness multiple CPUs and a GPU in the context of irregular numerical codes. Through multiple experiments on an optimized Sparse LU implementation, we showed that multigranularity improves heterogeneous scheduling, increasing spatial locality and leading to a better GPU utilization. We also demonstrated that scheduling should be guided by the software cache to amplify temporal locality, eliminating costly data transfers more effectively. We are working on adding multi-GPU support, relying on a distributed shared memory mechanism such as the one used in StarPU. We also plan to manage data dependences more accurately to exploit more parallelism and to hide the latency of sequential parts.

References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proc. of the 12th ACM Symp. on Parallel Algorithms and Architectures (2000)
2. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., Tomov, S.: A hybridization methodology for high-performance linear algebra software for GPUs. In: GPU Computing Gems. Morgan Kaufmann (2010)
3. Amestoy, P.R., Duff, I.S., L'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J. Matrix Anal. Appl. 23, 15–41 (2001)
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In: Euro-Par'09. pp. 863–874 (2009)
5. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An extension of the StarSs programming model for platforms with multiple GPUs. In: Euro-Par'09
6. Baboulin, M., Dongarra, J., Herrmann, J., Tomov, S.: Accelerating linear system solutions using randomization techniques. Tech. rep., INRIA (2011)
7. Banino, C., Beaumont, O., Carter, L., Ferrante, J., Legrand, A., Robert, Y.: Scheduling strategies for master-slave tasking on heterogeneous processor platforms. IEEE TPDS (2004)
8. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. In: Euro-Par'08. pp. 739–748. Springer-Verlag (2008)
9. Bellens, P., Perez, J.M., Cabarcas, F., Ramirez, A., Badia, R.M., Labarta, J.: CellSs: Scheduling techniques to better exploit memory hierarchy. Sci. Program. 17, 77–95 (2009)
10. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46, 720–748 (1999)
11. Deisher, M., Smelyanskiy, M., Nickerson, B., Lee, V.W., Chuvelev, M., Dubey, P.: Designing and dynamically load balancing hybrid LU for multi/many-core. In: ISC (2011)
12. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid multi-core parallel programming environment. In: Proc. of the Workshop on GPGPU'07 (2007)
13. Faverge, M., Lacoste, X., Ramet, P.: A NUMA aware scheduler for a parallel sparse direct solver. In: PMAA'08 (2008)
14. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO'07 (2007)
15. Gautier, T., Roch, J.L., Wagner, F.: Fine grain distributed implementation of a dataflow language with provable performances. In: Proc. of the 7th Int. Conference on Computational Science, Part II. pp. 593–600. Springer-Verlag (2007)
16. Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., Hwu, W.m.W.: An asymmetric distributed shared memory model for heterogeneous parallel systems. In: Proc. of the 15th Edition of ASPLOS. pp. 347–358. ACM (2010)
17. Hermann, E., Raffin, B., Faure, F., Gautier, T., Allard, J.: Multi-GPU and multi-CPU parallelization for interactive physics simulations. In: Euro-Par'10. pp. 235–246 (2010)
18. Jiménez, V.J., Vilanova, L., Gelado, I., Gil, M., Fursin, G., Navarro, N.: Predictive runtime code scheduling for heterogeneous architectures. In: HiPEAC'09. pp. 19–33 (2009)
19. Leung, A., Lhoták, O., Lashari, G.: Automatic parallelization for graphics processing units. In: PPPJ'09. pp. 91–100. ACM (2009)
20. Li, X.S.: An overview of SuperLU: Algorithms, implementation, and user interface. ACM Trans. Math. Softw. 31, 302–325 (2005)
21. Quintin, J.N., Wagner, F.: Hierarchical work-stealing. Euro-Par'10 (2010)
22. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid gpu accelerated manycore systems. Parallel Comput. 36, 232–240 (2010)
23. Volkov, V., Demmel, J.: LU, QR and cholesky factorizations using vector capabilities of GPUs. Tech. rep., University of California, Berkeley (2008)